HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Introduction to Object-Oriented Programming Concepts (OOP)

http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

Nirosh L.W.C. (MBA, UOC), Version 29. September 2017, adapted for CAS FEE by Silvan Gehrig

Introduction

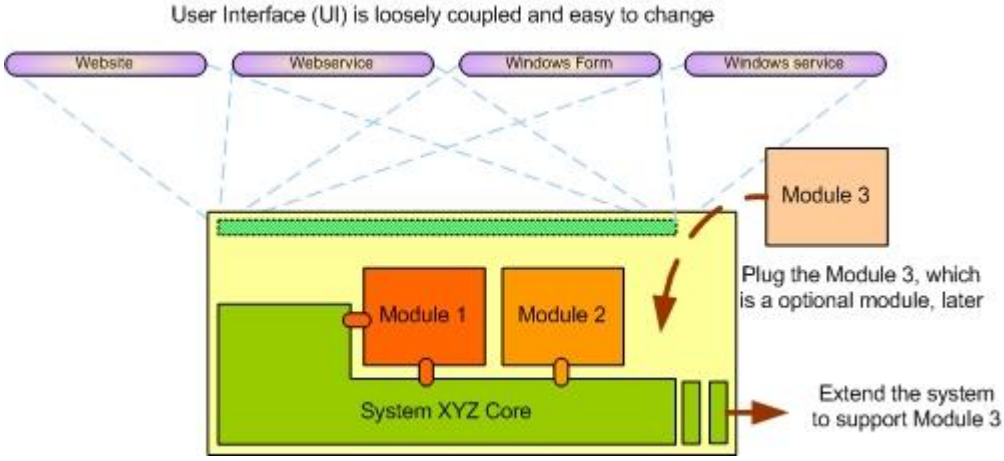
This article is an effort to provide an accurate information pool for new developers on the basics of software architecture, focusing on **O**bject-**O**riented **P**rogramming (OOP).

What is Software Architecture?

Software architecture is defined to be the rules, heuristics, and patterns governing:

- Partitioning the problem and the system to be built into discrete pieces.
- Techniques used to create interfaces between these pieces.
- Techniques used to manage overall structure and flow.
- Techniques used to interface the system to its environment.
- Appropriate use of development and delivery approaches, techniques and tools.

Why Architecture is Important?



The primary goal of software architecture is to define the non-functional requirements of a system and define the environment. The detailed design is followed by a definition of how to deliver the functional behavior within the architectural rules. Architecture is important because it:

- Controls complexity
- Enforces best practices
- Gives consistency and uniformity
- Increases predictability
- Enables **re-use** and **testing**.

What is OOP?

OOP is a design philosophy. It stands for **O**bject-**O**riented **P**rogramming. Object-Oriented Programming uses a different set of programming languages than old procedural programming languages (C, Pascal, etc.). Everything in OOP is grouped as self-sustainable "objects". Hence, you gain reusability by means of four main object-oriented programming concepts.

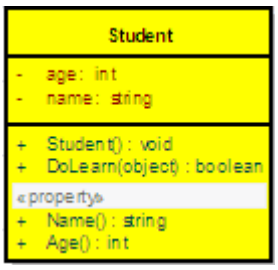
In order to clearly understand the object orientation model, let's take your "hand" as an example. The "hand" is a class. Your body has two objects of the type "hand", named "left hand" and "right hand". Their main functions are controlled or managed by a set of electrical signals sent through your shoulders (through an interface). So the shoulder is an interface that your body uses to interact with your hands. The hand is a well-architected class. The hand is being reused to create the left hand and the right hand by slightly changing the properties of it.

What is an Object?

An object can be considered a "thing" that can perform a set of related activities. The set of activities that the object performs defines the object's behavior. For example, the Hand (object) can grip something, or a Student (object) can give their name or address.

In pure OOP terms an object is an instance of a class.

What is a Class?



A class is simply a representation of a type of object (see picture above). It is the blueprint (or plan, or template) that describes the details of an object. A class is the blueprint from which the individual objects are created. Class is composed of three things: a name (Student), attributes (age, name), and operations (DoLearn).

```
function Student { }                                    // ES5 Syntax

class Student { }                                       // ES 2015 / TypeScript Syntax
```

According to the sample given below we can say that the Student object, named objectStudent, has been created out of the Student class.

```
const objectStudent = new Student();
```

In real world, you'll often find many individual objects all of the same kind. As an example, there may be thousands of other bicycles in existence, all of the same make and model. Each bicycle has built from the same blueprint. In object-oriented terms, we say that the bicycle is an instance of the class of objects known as bicycles.

In the software world, though you may not have realized it, you have already used classes. For example, the *TextBox* control is made out of the *TextBox* class, which defines its appearance and capabilities. Each time you drag a *TextBox* control, you are actually creating a new instance of the *TextBox* class.

How to identify and design a Class?

This is an art; each designer uses different techniques to identify classes. However according to Object-Oriented Design Principles, there are five principles that you must follow when designing a class:

- SRP - The Single Responsibility Principle
  A class should have one, and only one, reason to change.
- OCP - The Open Closed Principle
  Should be able to extend any classes' behaviors, without modifying the classes.
- LSP - The Liskov Substitution Principle
  Derived classes must be substitutable for their base classes.
- DIP - The Dependency Inversion Principle
  Depend on abstractions, not on concretions.
- ISP - The Interface Segregation Principle
  Make fine grained interfaces that are client specific.

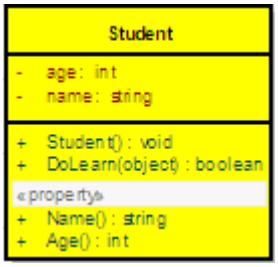
A software system may consist of many classes. When you have many classes, they need to be managed. Think of a big organization, its work force exceeding several thousand employees (let's take one employee as one class). In order to manage such a work force, you need to have proper management policies in place. Same technique can be applied to manage classes of your software system. In order to manage the classes of a software system, and to reduce the complexity, system designers use several techniques, which can be grouped under four main concepts named

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism.

These concepts are the four central points of OOP world and in software term, they are called four main Object-Oriented Programming (OOP) concepts.

What is Encapsulation (or Information Hiding)?

The encapsulation is the inclusion of all the resources needed for the object to function, basically, the methods and the data. In OOP the encapsulation is mainly achieved by creating classes, the classes expose public methods and properties. A class is kind of a container or capsule or a cell, which encapsulate a set of methods, attribute and properties to provide its indented functionalities to other classes. In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the system. That idea of encapsulation is to hide how a class does its business, while allowing other classes to make requests of it.



In order to modularize / define the functionality of a class, it can use functions or properties exposed by another class in many different ways. According to Object-Oriented Programming there are several techniques classes can link with each other. Those techniques are named association, aggregation, and composition.

There are several other ways an encapsulation can be used. To substantiate the techniques mentioned above, we can take the usage of `#private` members (EcmaScript TC39 proposal / Stage 3) and EcmaScript 2015 modules to hide information and functionality from external code.

What is the difference between a Class and an Interface?

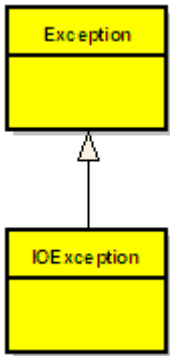
A class can be defined to implement an interface. When a class implements an interface, an object of such class inherits all public functions and methods.

A class and an interface are two different types (conceptually). Theoretically, a class emphasis the idea of encapsulation while an interface emphasis the idea of abstraction (by suppressing the details of the implementation).

Remarks: Interfaces are not natively supported in JavaScript (ES5 / ES 2015) – the programmer has to construct interface classes by defining functions/classes with empty implementations and inherit from them.

What is Inheritance?

The ability of a new class to be created, from an existing class by extending it, is called inheritance.



According to the example above the class (IOException) which is called the derived class or subclass inherits the members of an existing class (Exception). It's called the base class or super-class. The class IOException can extend the functionality of the class Exception by adding new types and methods and by overriding existing ones.

The inheritance is closely related with specialization. It is important to discuss the concepts together with generalization to better understand and to reduce the complexity.

One of the most important relationships among objects in the real world is specialization, which can be described as the "is-a" relationship. When we say that a dog is a mammal, we mean that the dog is a specialized kind of mammal. It has all the characteristics of any mammal (it bears live young, nurses with milk, has hair), but it specializes these characteristics to the familiar characteristics of canis domesticus. A cat is also a mammal. As such, we expect it to share certain characteristics with the dog that are generalized in Mammal, but to differ in those characteristics that are specialized in cats.

The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization coin: Mammal generalizes what is common between dogs and cats, and dogs and cats specialize mammals to their own specific subtypes.

In OOP, the specialization relationship is implemented using the principle called inheritance. This is the most common and most natural and widely accepted way of implement this relationship.

What is Polymorphism?

Polymorphisms is a generic term that means 'many shapes'. More precisely Polymorphisms means the ability to request that the same operations be performed by a wide range of different types of things.

At times, I used to think that understanding Object-Oriented Programming concepts have made it difficult since they have grouped under four main concepts, while each concept is closely related with one another. Hence one has to be extremely careful to correctly understand each concept separately, while understanding the way each related with other concepts.

In OOP the polymorphisms is achieved by using many different techniques named method overloading, operator overloading, and method overriding.

What is Method Overloading?

Method overloading is the ability to define several methods all with the same name.

```
function MyLogger                                          // ES5 Syntax
{
    this.logError = function(errorData)
    {
        if (errorData instanceof Exception) {
            // Implementation for logger.logError(new Exception()) goes here
        }
        else if (errorData instanceof String) {
            // Implementation for logger.logError("Error Message") goes here
        }
    }
}
```

```
class MyLogger                                             // ES 2015 / TypeScript Syntax
{
    logError(errorData) // pass Exception or String as errorData
    {
        if (errorData instanceof Exception) {
            // Implementation for logger.logError(new Exception()) goes here
        }
        else if (errorData instanceof String) {
            // Implementation for logger.logError("Error Message") goes here
        }
    }
}
```

Remarks: Method Overloading isn't natively supported in JavaScript (ES5 / ES 2015) – the programmer has to construct the support for arguments with different types manually.

What is Operator Overloading?

The operator overloading (less commonly known as ad-hoc polymorphisms) is a specific case of polymorphisms in which some or all of operators like +, - or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments.

```
function Square                                          // ES5 Syntax
{
    this.length = 5;
    this.valueOf = function() // override Object.valueOf()
    {
        return this.length * this.length; // return primitive JavaScript type here
    }
}

class Square                                             // ES 2015 / TypeScript Syntax
{
    constructor()
    {
        this.length = 5;
    }
    valueOf() // override Object.valueOf()
    {
        return this.length * this.length; // return primitive JavaScript type here
    }
}
```

In the example above I have overloaded the valueOf() operator for returning the area of the Square. The JavaScript interpreter uses the valueOf() method when applying operators such as +, - or ==.

What is Method Overriding?

Method overriding is a feature that allows a subclass to override a specific implementation of a method that is already provided by one of its super-classes.

A subclass can give its own definition of methods but need to have the same signature as the method in its super-class. This means when overriding a method, the subclass's method should implement same name and parameter list as the super-class's overridden method.

Remarks: Method Overriding must be applied in different ways when using JavaScript. The applied implementation strongly depends on the used inheritance mechanism.

What is a Client – Server (two-tier) Architecture?

The two-tier architecture refers to client/server architectures as well. The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s, and later it was adapted to World Wide Web programming.

According to the modern days use of two-tier architecture the user interfaces (web pages) run on the client and the database is usual stored on the server. The actual application logic can run on either the client (by using browsers such as Chrome, Firefox, Spartan, etc.) or the web-server (by using IIS, Apache, node.js, etc.). If the application is running on the server, the user interface's logic can directly access the database; otherwise additional APIs are required to access the data behind web-servers.